COURSE NOTES FOR CPSC 448

DIRECTED STUDIES IN COMPUTATIONAL GEOMETRY

Jean-Sébastien Basque-Girouard
University of British-Columbia

Last Updated

June 29, 2013

Contents

Introduction

These are course notes for an undergraduate directed studies course in computational geometry under professor David Kirkpatrick's supervision at the University of British-Columbia for the summer of 2013. The course initially follows the notes of David Mount [1], and uses [2] as its textbook. The appendix includes materials that were used or mentioned in the notes and for which, when they weren't entirely new to me, I wanted to solidify my understanding. While this document is under construction, this appendix may include a number of things that will turn out to be irrelevant in the body of the notes. Please inform me of any mistakes at js.b.girouard at google's email service.

Chapter 1

Convex Hulls

Definition 1.1. A set C is said to be *convex* if for any $x, y \in C$ and $\theta \in [0,1]$, $\theta x + (1-\theta)y \in C$. That is, for $x, y \in C$, the line segment between x and y is also in C.

Definition 1.2. The *convex hull* of a set of points S is the smallest convex set that contains S.

There is a number of characterizations for this definition; see Appendix A.

1.1 Algorithms

We present a few algorithms for computing the convex hull of a set of points.

1.1.1 Graham's Scan (Andrew's Algorithm)

In this algorithm, we sort the points in order of their x coordinates, and then proceed incrementally from left to right to find the upper hull, and then from right to left to find the lower hull, and we concatenate the results.

For each point p_i that we add, the verify that the points (p_{i-2}, p_{i-1}, p_i) form a right turn. If they do, we add p_i to the upper hull and repeat with the next point. Otherwise we remove the middle point, and test the orientation of (p_{i-3}, p_{i-2}, p_i). We proceed until we reach the rightmost point. The lower hull is constructed symmtrically, and the two hulls are then combined to form the convex hull.

Lemma 1.3. *We can compute the orientation of three points a, b, and c by taking the determinant*

$$\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{vmatrix}.$$

The points are clockwise if this is negative, counterclockwise if this is positive, and colinear if this is equal to 0.

Time Complexity

Graham's scan has a cost in $O(n \log n)$. We must first sort each item in $O(n \log n)$ time, and then test the triples for orientation, removing points that are not in the hull. At first sight, it may seem, since we could remove almost every point in the hull on any given step, that this would overall be in $O(n^2)$. While this is certainly true, this upper bound is not tight. Notice that by taking an amortized point of view, we see that any point can only be removed once from the hull being constructed. It follows that this step runs in $O(n)$ time. Therefore, the algorithm is in $O(n \log n)$.

1.1.2 Divide and Conquer

Here we do something similar to the mergesort algorithm. Indeed, many sorting algorithms can be easily transformed into convex hull algorithms. In this case, we recursively find the convex hull of two subsets of the points, and merge them by finding their outer tangents.

We begin again by sorting the points in order of their x values. We then separate the points into to halves, and compute the convex hull by recursively running the algorithm on each half of the points, and merge the two hulls together. If our set of points has size 3 or less, we have the hull in constant time, since every point is in the hull.

In order to merge two hulls, we must find the outer tangents. The way we do this is by picking the rightmost point a in the left hull, the leftmost point b in the right hull, and going in a counterclockwise order for the right hull, we find the orientation of the triple a, b, successor of b. If the orientation is positive (counterclockwise), then we test the orientation of the triple consisting of the predecessor of a,a,b in the same manner. Otherwise, we move along the right hull until the predicate is satisfied. Thus we will move back and forth from the right to the left hull, until we have the lower tangent between the two hulls. The upper tangent is found symmetrically.

Time Complexity

This algorithm also runs in $O(n \log n)$. The sorting step takes as long. What we need to show is that finding the tangents does not exceed this cost. Using an amortized analysis, we can note that each point in the set can only be discarded once from being on the hull of a subset. Therefore, this step takes $O(m)$ time, where m is the number of points on the two subhulls. Noting that $m \leq n$, we are satisfied.

1.1.3 Quickhull

This algorithm has, as its name suggests, an approach similar to quicksort. We find the minimum and maximum x values in our set of points, and use them to define a straight line. We then find the point above the line that is furthest from it, and connect it to the ends of our line, forming two more lines on which the algorithm is repeated recursively, until the upper hull is found. We proceed symmetrically to find the lower hull.

Time Complexity

Given a set of uniformly distrubuted inputs, the algorithm filters half of the remaining points from being candidates before it recurses. However, when given certain inputs, like a circle, the algorithm will filter out no points on any of the recursive calls. Consequently, in the worst case, the algorithm runs in $O(n^2)$ time. Thus the running time of quickhull depends on the shape of the input, in the same way as quicksort is inefficient for an already sorted list. Unfortunately, unlike quicksort, this cannot be remedied with randomization. Thus one can speak of the average running time of the algorithm being in $O(n \log n)$, but this does not make away with the worst case analysis.

1.1.4 Jarvis's March

It is clear that the convex hull of a set of points is at most as large as the set of points itself, and typically smaller. Jarvis's march takes advantage of that by computing in the hull with an output-sensitive cost.

Start with the point p_1 with the lowest y value, and assume that there is a point $p_0 = (-\infty, 0)$. Then find the point q that maximizes the angle $\angle p_0, p_1, q$. That point is the next vertex in the convex hull. Repeat the procedure for each point added to the hull until we reach p_1 again.

Time Complexity

Finding the lowest y value can be done in $O(n)$ time. Finding the max of $\angle p_{k-1}, p_k q$ can also be done in $O(n)$ time, and we do this a total of h times, where h is the number of points on the hull. Therefore, Jarvis's march has a cost in $O(nh)$. If $h < \log n$, this algorithm has better performance than Graham's scan or our divide and conquer algorithm.

1.1.5 Kirkpatrick-Seidel Algorithm

Continuing with output-sensitive algorithms, we have the Kirkptrick-Seidel algorithm. In this algorithm, we succeed in improving the running time to $n \log h$, where h is the number of items on the hull.

The way it works is by using a variation of the divide and conquer paradigm called *marriage before conquest*, where elements of the input that are known not to belong to the output are eliminated before the merge step, improving its performance. The algorithm finds the upper hull by first finding the median x value, L. It then finds the *bridge* that crosses the line L, that is, the edge of the hull that intersects it. It can then eliminate any point below the bridge from consideration, and recursively acts on the two separated sets of points that are remaining after the operation.

The way in which we can determine the bridge is by first making the points into $\lfloor \frac{n}{2} \rfloor$ disjoint pairs. Then, we note that the slope of the bridge is less than the slope of any supporting line defined by two points that are both left of L, and greater than the slope of any supporting line defined by two points that are both right of L. For each pair, we calculate the slope of the line that they define. We then determine the median slope K, and classify each pair according to whether the slope that they give is less than, equal to, or greater than the median slope. Next, we find the points that maximize $y - Kx$. This will give us the points that form a line that is closest to the slope K, and most likely to be the bridge. So we take the leftmost point of those, and the rightmost point. If they are on each side of L, then we have found our bridge. Otherwise, we recursively perform this bridge-finding procedure on a subset of the points that contains the points that are possible candidates for the bridge.

The candidates for the recursive call will be the left point of any pair that was classified as having a slope less than K while being to the right of L, the right point of any pair with a slope greater than K while being to the left or on L, as well as any pair on the left of L with slope smaller than K and any pair on the right of L with a slope greater than K.

It should be noted that the bridge-finding algorithm is a linear programming problem in 2 dimensions.

Time Complexity

As we pointed out, this algorithm runs in $O(n \log h)$ time. We first find the left and right end points in $O(n)$. Then connecting the components is done recursively by eliminating any point that is below the bridge and calling the procedure on the two sets of remaining points to the left and right of the bridge. Thus the recursion is on at most $\frac{n}{2}$ points each, with h_l hull points on the left and h_r hull points on the right. Using calculus we can see that the maximum is achieved when $h_l = h_r = \frac{h}{2}$. This recurrence solves to $O(n \log h)$.

The bridge-finding algorithm runs in linear time, and each recursive call is done on at most $\frac{3}{4}$ of the original number of points. We thus get the recurrence

$$T(n) = T\left(\frac{3}{4}n\right) + O(n)$$

for $n > 2$. This solves to $O(n)$ since this recurrence forms a geometric series.

1.1.6 Chan's Algorithm

Chan's algorithm works by performing Graham's scan on subsets of size h^*, where $h \leq h^* \leq h^2$. We then merge the smaller convex hulls into the final result using Jarvis's march. The interesting thing to note here is that by combining two inferior algorithms, we get a complexity result that is actually better than either of them can achieve.

We proceed first by arbitrarily creating n/h^* subsets, each of size roughly h^*. We then merge the hulls by running Jarvis's algorithm on the hulls as if they were points. That is, we start with the lowest point p_1 on the lowest hull, and find the next point that maximizes the angle between p_0 (as defined earlier), p_1, and the next point q, which is either on the same hull or on another hull. We only need to look at the points for which the segment p_1q is a tangent. Again, we proceed until we reach p_1 again.

Time Complexity

Chan's algorithm runs in $O(n \log h)$ time. Running Graham's scan on each subset can be done in $O(h^* \log h^*)$ time. This is repeated n/h^* times, for a cost in $O(n \log h^*)$. What is important to show is that the merging step does not increase this cost. Note that the hulls of each subset are more structured than the original points, which is something we can exploit to find the tangents efficiently.

Lemma 1.4. *Consider a convex polygon K in the plane and a point p that is external to K, such that the vertices of K are stored in cyclic order in an array. Then the two tangents from p to K (more formally, the two supporting lines for K that pass through p) can each be computed in time O(log m), where m is the number of vertices of K.*

Proof. We will show how to find one tangent. The procedure can be repeated symmetrically to find the other tangent. Let $p_1, \ldots, p_m$ be the vertices of K. It is possible to perform a binary search on the vertices of K in the following manner. Pick the middle element of K, that is, let $k = \lfloor \frac{m}{2} \rfloor$ and pick p_k. Then compute the orientations of the two triples (p, p_{k-1}, p_k) and (p, p_k, p_{k+1}). If their signs differ, then (p, p_k) is a tangent. If the orientations are both positive, then recursively look for the tangent in the same manner in the points $p_1, \ldots, p_{k-1}$. If they are both negative, recursively perform the procedure on the points p_{k+1}, p_m. By the fact that binary search takes $O(\log m)$ time on a list of m elements, we get the desired result. ■

Thus we merge each hull in $O(\log h^*)$, for a total in $O(\frac{n}{h^*} \log h^*)$. Since we have at most h^* such steps, it follows that we have our total running time in $O(n \log h^*)$. Given that h^* is no more than h^2, this is equivalent to $O(n \log h)$.

Now, in order to find the optimal value of h^*, we perform the algorithm successively on subsets of size 2, 4, 16, 256, etc., letting the algorithm fail if $h^* < h$. That is, we square the size of the last estimate each time, until the algorithm succeeds. Then we are sure to get subsets of the correct size eventually, and some analysis can show that the number of values that we try will at most double the cost of the algorithm. Indeed, note that when the algorithm succeeds, we are certain that $h^* \in [h, h^2)$. What we must show is that running the algorithm on smaller sizes of h^* does not increase our asymptotic cost. Note that we will run the algorithm at most $\lg \lg h$ times, at a cost in $O(n \log h_i^*)$ for each attempt i. Now, $h_i^* = 2^{2^i}$. Therefore, each attempt's cost is in $O(n 2^i)$. Thus our cost function, up to a constant factor, is the following:

$$\begin{aligned}
T(n,h) &= \sum_{i=1}^{\lg \lg h} n 2^i = n \sum_{i=1}^{\lg \lg h} 2^i \\
&= n \frac{2^{\lg \lg h + 1} - 1}{2 - 1} && \text{(by geometric series)} \\
&= n(2 \cdot 2^{\lg \lg h} - 1) = n(2 \lg h - 1) \\
&= 2n \lg h - n \\
&\in O(n \log h).
\end{aligned}$$

We thus get the desired result.

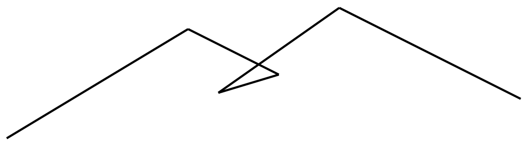
1.1.7 Convex Hull of a Polygon

Given a polygon, there are algorithms for finding its convex hull that run in linear time.

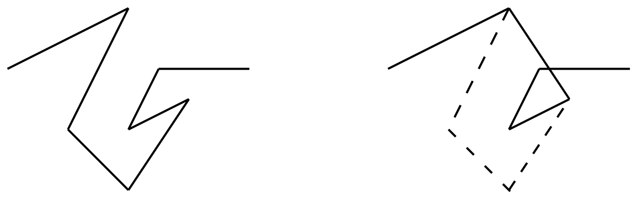
Definition 1.5. A *polygonal chain* P is a curve represented by an ordered tuple $P = (p_0, p_1, ..., p_n)$, where each p_i, called a *vertex*, is a point in space.

Of the algorithms that we have seen so far, Graham's scan would appear to be ideal here, in that we could achieve linear time by avoiding the sorting step. We will see, however, that this is not the case.

Notice that Graham's scan would fail in the case of a self-intersecting polygonal chain. Indeed, the following figure shows a polygonal chain with no left turns for which Graham's scan would return an incorect shape.



The same problem can occur in a non-self-intersecting chain since Graham's scan could produce an intersection.



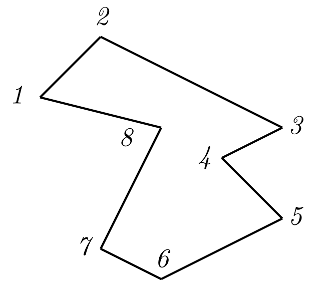
Thus, we need a different method. One that solves this problem is Melkman's algorithm. It is better described with pseudocode. We maintain a deque $D = \langle d_b, d_{b+1}, \ldots, d_{t-1}, d_t \rangle$ with the indices representing the convex hull of the points examined up to that point. The value d_t stands for the top of the deque, and the value d_b stands for the bottom of the deque. Popping from D means to make $t \leftarrow t-1$, and removing from D means to make $b \leftarrow b+1$. Pushing is done on top, inserting is done at the bottom.

Algorithm 1 Melkman($v_1, \ldots, v_n$)

```
if orient($v_1, v_2, v_3$) = left then
    $D \leftarrow \langle 3, 1, 2, 3 \rangle$
else
    $D \leftarrow \langle 3, 2, 1, 3 \rangle$
end if
for $i \leftarrow 4$ to $n$ do
    while orient($d_{t-1}, d_t, v_i$) = left and orient($d_b, d_{b+1}, v_i$) = left do
        $i \leftarrow i + 1$
    end while
    while orient($d_{t-1}, d_t, v_i$) $\neq$ left do
        pop($d_t$)
    end while
    push($v_i$)
    while orient($v_i, d_b, d_{b+1}$) $\neq$ left do
        remove($d_b$)
    end while
    insert($v_i$)
end for
```

Assume that the algorithm stops when we finish treating the last vertex. Given that the deque comprises the convex hull of a subset of the points at any given iteration, we can use an inductive argument to prove the correctness of the algorithm.

Example 1.6. *Suppose that we have a polygon $P = \langle (0,8), (2,10), (8,7), (6,6), (8,4), (4,2), (2,3), (4,7) \rangle$.*



Let us run through the algorithm step by step. The first three vertices form a right turn, therefore we start with $D = \langle 3, 2, 1, 3 \rangle$.
First iteration of the `for` *loop. We let $i \leftarrow 4$. The first while loop gets skipped, and the second one pops once before pushing the new vertex, 4. We get $D = \langle 3, 2, 1, 4 \rangle$. The last while loop is skipped, and we insert 4. $D = \langle 4, 3, 2, 1, 4 \rangle$.*
Second iteration. Let $i \leftarrow 5$. We skip the first while loop again. After the second one, we've popped 4 and pushed 5. In the last while loop, we remove 4; we then insert 5. $D = \langle 5, 3, 2, 1, 5 \rangle$.
Third iteration. Let $i \leftarrow 6$. Skip the first while loop. In the second one, we pop 5; we then push 6. The last iteration is skipped, and we insert 6. $D = \langle 6, 5, 3, 2, 1, 6 \rangle$.
Fourth iteration. Let $i \leftarrow 7$. The first while loop is skipped. The second one pops 6; we push 7. The last iteration is skipped, we insert 7. $D = \langle 7, 6, 5, 3, 2, 1, 7 \rangle$.
Fifth and last iteration. Let $i \leftarrow 8$. The body of the first while loop is executed, because vertex 8 is inside the current hull. Since we have finished treating the last vertex, we are done.

Time Complexity

We can see from the algorithm that any point in the set of vertices will be added or removed from the deque at most once. Therefore the Melkman's algorithm runs in linear time, as claimed.

1.2 Degeneracy

So far, we have been assuming that no three points are colinear. We can deal with degeneracies such as this with symbolic pertrubations of the inputs.

1.3 Optimality

As it turns out, the $O(n \log h)$ bound is optimal for convex hull algorithms. We will argue this by showing that the convex hull size verification problem must take $\Omega(n \log h)$ steps, where n and h are as before. Certainly, if verifying the size of a convex hull takes this amount of steps, then finding the hull itself must take at least as many.

1.3.1 Decision Trees

Definition 1.7. By a *linear decision tree*, we mean a decision tree for which each node represents a vector $(a_0, \ldots, a_n)$ that, for an input $(x_1, \ldots, x_n)$, describes a hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = -a_0$. Each node's children stand for $<$, $>$, and $=$, in relation to 0.

Proposition 1.8. *If any two inputs in a linear decision tree reach the same leaf, then they must be part of the same connected component in the input space.*

Proof. Note that any node in a linear decision tree indicates whether the input is in one of two halfspaces, or on a halfplane defined by the test. Hence an input reaching a leaf draws a convex polyhedron in the input space. It can be shown that convex sets are connected. Therefore, every input that goes to one leaf belongs to the same connected component in the input space. ■

Theorem 1.9 (Ben-Or)**.** *Let $W \subset R^n$ be any set, and let T be a computation tree that solves the membership problem for W. If N is the number of disjoint connected components of W, and h is the height of T, then*

$$2^h 3^{n+h} \geq N.$$

In other words, $h \geq \frac{\log N - n \log 3}{1 + \log 3} \in \Omega(\log N - n)$.

Proposition 1.10. *Solving the multiset size verification problem ("does a multiset have k distinct elements?") takes $\Omega(n \log k)$ time for an input of size n.*

Proof. Let $\mathbf{z} \in \mathbb{R}^n$ be our multiset. Using a counting argument, we note that there are $k!$ ways to arrange the k distinct components of $\mathbf{z}$, and that there are k^{n-k} ways to arrange the remaining ones. Thus there are $k!k^{n-k}$ possible permutations of the components. It follows that any decision tree that decides this problem must have $k!k^{n-k}$ leaves, corresponding to the same number of connected components in the input space. Using Theorem 1.9, we get that the height of the computation tree for solving the multiset size verification problem is in

$$\begin{aligned}\Omega(\log(k!k^{n-k}) - n) &= \Omega(k \log k + (n-k)\log k - n)\\ &= \Omega(k \log k + n \log k - k \log k - n)\\ &= \Omega(n \log k - n) = \Omega(n \log k),\end{aligned}$$

thus proving the result. ■

Theorem 1.11. *The convex hull size verification problem takes $\Omega(n \log h)$ steps to compute.*

Proof. We can provide a reduction from the multiset size verification problem to the convex hull size verification problem, showing that it is also in $\Omega(n \log k)$, where $k = h$. Suppose that $\mathbf{z} \in \mathbb{R}^n$ and k are an instance of the multiset size verification problem. Let $P = \{p_1, \ldots, p_n\}$ be a set of points, where $p_i = (z_i, z_i^2) \in \mathbb{R}^2$. Then the convex hull of P has k elements if and only if $\mathbf{z}$ has k distinct elements. ■

Chapter 2

Linear Programming

2.1 An Incremental Algorithm

Linear programs in spaces of constant dimension can be solved somewhat efficiently using an incremental algorithm. Let d be the number of dimensions (variables) of the problem. We will construct a solution by adding constraints one at a time, updating the optimal vertex if necessary.

Suppose that we have d points forming a vertex from which to start. This is our first optimal vertex. By adding a constraint, we either have that the current optimal vertex satisfies it, in which case it remains the optimal vertex, or it does not, in which case we need to update the optimal vertex. The way we do this is by projecting the objective function onto the hyperplane defined by the new constraint. We then perform this along the hyperplane, that is, on a space of $d - 1$ dimensions. We recursively do this until we have 1 dimension, that is, the hyperplane is a line. At that point, all that is left to do is find the first intersection to the line in the direction of the objective function, which can be done in $O(i)$ time, where i is the number of constraints added so far.

2.1.1 Backwards Analysis

The above algorithm runs in

$$\sum_{i=1}^{n} O(i) \in O(n^2)$$

in the worst case.

However, this running time assumes that we must update the new optimal vertex every time we add a constraint. If we assume some uniform distribution for the order in which we add constraints, this case is very unlikely: $\frac{1}{n!}$ is its probability. Consequently, it makes sense to randomize the algorithm to take advantage of this, and improve its *expected* running time.

To analyze the cost of this randomized version, we use a technique called *backwards analysis*. The way it works is by noting that after we've added the i^{th} constraint, the optimal vertex is updated only if this constraint was one of the hyperplanes that defines it. Since they are uniformly distributed, this constraint has probability $\frac{1}{i-d}$ of being in any of the $i-d$ positions from $d+1$ to i. Thus, for constraint i, the probability that we updated the optimal vertex upon its addition is $\frac{d}{i-d}$. Thus we get the expected cost

$$T(d,n) = \sum_{i=d+1}^{n} \left[\frac{d}{i-d} \left(O(di) + T(d-1, i-1) \right) \right] \in O(n)$$

for constant d (note that the constant is actually $d!$, so the assumption that d is constant is significant).

The rest of the algorithm clearly runs in linear time, as the situation where we do not need to update the optimum can be taken care of in constant time (that is, in $O(d)$).

2.1.2 LP-type Problems

Many optimization problems can be solved in a manner similar to the linear programming solution above. More precisely, any optimization problem that can be solved incrementally, where each addition is either part of the current solution, or defines a lower-dimensional problem, can be solved using this technique.

Smallest Enclosing Disk

The smallest enclosing disk problem asks to find the smallest circular closed disk that includes a set of n points. Even though this cannot be cast as a linear program, we can solve this with a similar randomized incremental procedure.

Thus we find the smallest enclosing disk for three of the points, and then add points one at a time, updating the disk if they fall outside of the current one. Using randomization as we did before, we can limit the expected number of updates to the current disk.

The way in which we update the disk is by finding the smallest enclosing disk of all the points seen so far, with the requirement that it include on its boundary the last point added. We then recurse in a way in which, when we encounter a point that lies outside of the current disk, we require that both this point and the previous point be on the boundary of the smallest enclosing disk. If we make it ones step further in the recursion, we require three points to be on the boundary, at which point we trivially have the smallest enclosing disk.

We can show that if the last point added falls outside of the current disk, then it must fall on the boundary of the updated disk. First note that if three points define the boundary of a smallest enclosing disk, then the angle of the arc between each point is at most π. Otherwise, the two points with angle greater than π between them could define a smaller circle whose diameter is the distance between the two points. Now, suppose that the newly added point does not fall on the boundary of the new disk. Then the new disk is defined by points from the previous disk. However, the intersection of the new bigger disk's boundary with the smaller previous disk must have an arc angle less than π. Hence two of the points defining the new disk must have angle greater than π between them. It follows that the new disk is not a smallest enclosing disk.

Convex Hulls in Higher Dimensions

We have seen that convex hull algorithms can benefit from being given a sorted list of points, as in Melkman's algorithm, in order to have a faster running time. However, can we profit from an ordering in higher dimensions? After all, the set of points is linear regardless of which dimension the points are in. Suppose that we have a set of points in 3-dimensional space. Which order can we exploit to find the hull quickly? It turns out that we cannot gain from an ordering. Still, efficient algorithms do exist in higher dimensions. Particularly, we can find the convex hull with a randomized incremental algorithm similar to what we have been seeing so far.

For each point that we add that is outside the current hull, we connect the vertices of the current hull that are on the boundary of what the new point can "see." That is, we want to find the hyperplanes that are tangent to the current hull, and that contain the new point. This is a linear programming problem. Once again, we only have to do this if the new point falls outside the current hull, suggesting randomization to get a similar analysis of the running time of the algorithm.

Star-Shaped Polygons

A simple polygon P is *star-shaped* if there is a point p in its interior such that for any point q on its boundary, the line segment between p and q is contained in P. We can test whether a polygon is star-shaped by adding the halfplanes that define the polygon one at a time, as in the linear programming solution above, using an arbitrary objective function. The polygon is star-shaped if the feasible region is not empty, and the "optimal" point that we have is our point p.

Chapter 3

Planar Point Location

Given a set of n line segments that define a planar subdivision and a point p, how can we tell where p is? One way is to divide the plane into vertical slabs that correspond to the x values of the endpoints of the line segments. The plane is then divided into regions that are trapezoids, triangles, or unbounded regions. When we find the region that includes p, we know in which region of the original subdivision p is. This can be done with two binary searches, one to find the correct slab, and one to find which section of that slab contains p. The $O(\log n)$ cost for searching is nice, but the data structure that we just defined takes $O(n^2)$ space. We would like to find a data structure that provides efficient query times as well as good space complexity.

3.1 Trapezoidal Maps

Provided with a set S of line segments, we pretend that they are surrounded by a box R so that no regions of interest are unbounded. We also assume that none of the segment end points share the same x value. We divide the area within R with a trapezoidal map: given a set of line segments in the plane, a trapezoidal map is constructed by shooting vertical "bullets" from each end point to form vertical line segments, called walls, that end when the bullet hits another line segment or a wall of a bounding box that we place around the set. Note that this map has at most $6n + 4$ vertices: three per line segment endpoint, and four for the corners of R. It also has at most $3n + 1$ trapezoids: remark that the left edge of each trapezoid is defined by exactly one point, which is either the left endpoint of a line segment (which can define the left edge of at most two trapezoids), the right endpoint of a line segment (which can define the left edge of one trapezoid), or the lower left corner of R.

We will create a search structure as part of the construction of the trapezoidal map. The search structure will be a directed acyclic graph where each leaf corresponds to a trapezoid in the map, and each internal node corresponds either to an endpoint (called an x-node) or a line segment (called a y-node).

Starting with the bounding box as the first trapezoid, which is denoted by a single leaf in the search structure, we add the line segments at random, shooting bullets from their endpoints and trimming any walls that they intersect. Each increment thus creates $K + 4$ new trapezoids, where K is the number of walls that the newly added segment intersects (4 created by the bullets shot from the end points, and one by each wall intersection). We must remove each trapezoid that the segment intersects from the map and the search structure. In order to find these, we perform a query on the current search structure to find the trapezoid in which the left endpoint of the line segment falls. From this, each intersected trapezoid will be to the right of the previous one, so we can traverse the structure in order until we reach the trapezoid where the right endpoint of the segment lies. Once we've done this, we can update the search structure by replacing each removed trapezoid by a tree with, at its leaves, the newly created subdivisions of the removed trapezoid.

The expected number of trapezoids created by the addition of a line segment is in $O(1)$: the intuition is that any long line segment that intersects many walls will shield future line segment additions from wall intersections. More rigourously, observe that each trapezoid is defined by at most four line segments.

Using a backwards analysis, we can note that after the addition of the i^{th} line segment, the probability that it affected any of the current trapezoids is $\frac{4}{i}$. Since the current map has $O(i)$ trapezoids, we get that the expected number of trapezoids created by the addition of a line segment is constant.

3.2 Degeneracy

Chapter 4

Voronoi Diagrams

Voronoi diagrams are a way of separating space into regions defined by a set of points P and a metric. Each region described by a point p corresponds to the set of points of space that are closer to p than other points in P. We call this the *Voronoi region around* p. We refer to points of P as sites.

Suppose that you find yourself in the 1990s, and you need to get to a payphone. A Voronoi diagram could be defined to tell you, given your current position, which payphone is the closest to you.

4.1 Fortune's Algorithm

This algorithm finds a Voronoi diagram for a set of points in the plane by sweeping a line across the points, and maintaining a sweep-line status that defines what is referred to as the *beach line*. The beach line is formed of parabolic curves that are monotone in the direction that the sweep line moves. The parabolic curves represent the points that are at an equal distance from a site and the sweep line, for sites that do not yet have a defined Voronoi region. We will assume here that the sweep line goes vertically, from top to bottom.

The sweep line consists of the ordered set of sites that make up the beach line. It is represented by a dictionary. As the sweep line moves, it will encounter new sites. When a site is discovered, we find the arc that is directly above it with a binary search where we find the points on the beach line that are directly left and directly right of it (make the simplifying assumption that new points do not fall directly below breakpoints). With these two points, we can find the breakpoint between them by finding the circle that passes through both sites and is tangent to the sweep line. Given the breakpoint, we know which arc lies directly above the new site. Suppose that this arc is given by point p_j. When we add the new point p_i directly under its arc, the dictionary changes from $\langle \dots, p_j, \dots \rangle$ to $\langle \dots, p_j, p_i, p_j, \dots \rangle$.

We also keep an event queue that holds all the y coordinates of the sites. We implement it as an ordered dictionary from which we can extract the point with the largest y coordinate. For each consecutive triple in the sweep line, we store in the event line the lowest y coordinate of the circle that is described by those three points.

Thus the algorithm works by following the sweep line and handling events as they are encountered. The events will be either sites or vertices of the Voronoi diagram.

4.2 Divide and Conquer Algorithm

This method of finding the Voronoi diagram proceeds by splitting the points into roughly two halves, left and right of the median x value. The Voronoi diagram of each side is then computed recursively, and we merge both of them by finding the bisector between each pair of points whose Voronoi regions intersect. The old Voronoi regions are then truncated at the bisector, resulting in the correct Voronoi diagram of the whole set of sites.

The union of the bisectors that we just mentioned is called the *contour*, and it represents the set of points that are at equal distances from one site on the left diagram and one site on the right diagram.

This algorithm generates the same recurrence as mergesort,

$$T(n) = 2T\left(\frac{n}{2}\right) + n,$$

which solves to $O(n \log n)$.

Chapter 5

Delaunay Triangulation

The *Delaunay triangulation* of a set of points is the dual structure of the Voronoi diagram. It represents the triangulation that we get when considering each site as a vertex, and each bisector of the Voronoi diagram as an edge between the two points that it separates. Note that in such a subdivision, the vertices of the Voronoi diagram will represent the faces of the Delaunay triangulation. Remember that the vertices have degree 3 when no four points are cocircular, hence the name *triangulation.* Indeed the term Delaunay *graph* is more appropriate in the general case. Still, given the general case, we can always triangulate the polygons that have greater than three sides arbitrarily to get a triangulation.

5.1 Properties

The Delaunay triangulation of a set of points P has a number of neat properties.

Convex Hull

Proposition 5.1. *The boundary of the Delaunay triangulation is also the boundary of the convex hull of P.*

Proof. Recall that the outer sites of the Voronoi diagram (which are also the boundary vertices of the Delaunay triangulation) define an unbounded Voronoi region. Assume that one boundary point p of the Delaunay triangulation is not on the convex hull. Further assume that this point is on the upper boundary. Then the circle defined by this point and the two hull points to its left and right has its centre outside the Delaunay boundary. This implies that there is a vertex of the Voronoi diagram outside the Delaunay triangulation. It follows that the Voronoi region of p is bounded. But this is a contradiction, since it is a boundary site in the Voronoi diagram. Therefore, the boundary of the Delaunay triangulation is also the boundary of the convex hull. ■

Empty Circumcircle

Proposition 5.2. *The circumcircle of the three points that form a triangle in the Delaunay triangulation of P is empty.*

Proof. The circle defined by the three points has its centre at the vertex of the Voronoi diagram. It follows that it is empty, since those three points are the closest points to this vertex. ■

Empty Circle

Proposition 5.3. *There is an empty circle that passes through any two sites if and only if the two sites are adjacent in the Delaunay triangulation.*

Proof. Let there be an empty circle passing through two sites. Then they must be neighbours in the Voronoi diagram. Hence they are adjacent in the Delaunay triangulation.

Now assume that two sites are adjacent in the Delaunay triangulation. By definition of the Delaunay triangulation, along with a third point with which they form a triangle, the circumcircle that they define has an empty interior. Thus there is an empty circle that passes through the two sites. ■

Closest Pair

Proposition 5.4. *The closest pair of points of P are adjacent in the Delaunay triangulation.*

Proof. Let i and j be the closest pair of points. Then there is an empty circle passing through i and j. It follows that they are neighbours in the Delaunay triangulation. ■

Minimum Spanning Tree

Theorem 5.5. *For weights equal to the Euclidean distance between to sites, the minimum spanning tree of P is a subgraph of the Delaunay triangulation of P.*

Proof. Suppose that the minimum spanning tree is not a subgraph of the Delaunay triangulation. Then there is at least one edge of the MST that is not an edge in the triangulation. Let the vertices of this edge be a and b. There is a point c in the circle whose diameter is defined by a and b. Observe that the distance from c to a or b is smaller than the distance between a and b. The point c belongs to another part of the MST. Suppose that we remove the edge between a and b. This results in two trees. Without loss of generality, assume that a and c are in the same tree. Add an edge from c to b. The result is a tree with smaller total weight than the original, since the edge between c and b is shorter than the edge that we removed between a and b. Hence the original tree was not the minimum spanning tree. It follows that the MST of P is a subgraph of the Delaunay triangulation of P. ■

Spanner

Definition 5.6. A t-spanner is a graph with the property that the distance between any two vertices is at most t times the Euclidean distance between them.

Theorem 5.7. *The Delaunay triangulation of P is a t-spanner, where $t = 4\pi\sqrt{3}/9$.*

Proof. TODO ■

Maximizing Angles and Edge Flipping

Theorem 5.8. *The Delaunay triangulation has the lexicographically largest angle sequence of all triangulations of P.*

Proof. TODO ■

5.2 Randomized Incremental Algorithm

Like convex hulls, we can find the Delaunay triangulation by many different methods. We will present a randomized incremental algorithm, which will allow us to use backwards analysis to assay its running time.

We start by creating three points that form a triangle large enough that it contains the set of points and the circumcircles that they define (we can do this symbolically by modifying the test of Lemma 5.9 to never show the vertices of the surrounding triangle to be in any circumcircle). We then add the sites in a random order, and with each new addition we update the triangulation.

Lemma 5.9. *For points a, b, c, and d, the determinant*

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix}$$

is positive if and only if d lies inside the circumcircle described by Δabc*, with a, b, and c in clockwise order.*

Proof. Observe that a point $p = (p_x, p_y)$ lies on a circle of radius r centred at (x, y) if and only if $(p_x - x)^2 + (p_y - y)^2 - r^2 = 0$. The determinant above is equal to

$$-d_x \underbrace{\begin{vmatrix} a_y & a_x^2 + a_y^2 & 1 \\ b_y & b_x^2 + b_y^2 & 1 \\ c_y & c_x^2 + c_y^2 & 1 \end{vmatrix}}_{A} + d_y \underbrace{\begin{vmatrix} a_x & a_x^2 + a_y^2 & 1 \\ b_x & b_x^2 + b_y^2 & 1 \\ c_x & c_x^2 + c_y^2 & 1 \end{vmatrix}}_{B} - (d_x^2 + d_y^2) \underbrace{\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}}_{C} + \underbrace{\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 \\ b_x & b_y & b_x^2 + b_y^2 \\ c_x & c_y & c_x^2 + c_y^2 \end{vmatrix}}_{D}.$$

Rewriting it more succinctly, this gives us

$$\begin{aligned} -Ad_x + Bd_y - C(d_x^2 + d_y^2) + D &= -C\left(\frac{A}{C}d_x - \frac{B}{C}d_y + d_x^2 + d_y^2 - \frac{D}{C}\right) \\ &= -C\left(\left(d_x + \frac{A}{2C}\right)^2 + \left(d_y - \frac{B}{2C}\right)^2 - \frac{A^2 + B^2}{4C^2} - \frac{D}{C}\right) \\ &= -C\left(\left(d_x + \frac{A}{2C}\right)^2 + \left(d_y - \frac{B}{2C}\right)^2 - \frac{A^2 + B^2 + 4CD}{4C^2}\right). \end{aligned}$$

Letting

$$r^2 = \frac{A^2 + B^2 + 4CD}{4C^2},$$

and noting that C is negative (see Lemma 1.3), we have that the determinant is positive if and only if the point d lies inside the circumcircle. TODO make this more clear ■

Remark 5.10. Observe that we can transform a circle $C = (x - a)^2 + (y - b)^2 - r^2$ into a plane by raising it to the unit three-dimensional paraboloid, using its parameters a, b, and r to define $C^* = a(x - a) + b(y - b) + r^2$. Similarly, we can take a point $p = (x, y)$ in two dimensional space into a point in three dimensional space $p^* = (x, y, x^2 + y^2)$, which will put it on the surface of the paraboloid. Then the point p lies inside the circle C if and only if the point p^* lies below the plane C^*.

When we add a new point p, we add edges from it to the three vertices of the triangle Δabc that contains it. We then test whether the three newly created triangles have empty circumcircles. To do this, we look at the vertices that are opposite of p across the edges of Δabc. If they fall in the circumcircles defined by p and a, b, c, then we must fix the triangulation by flipping the culpable edges of Δabc. This flipping procedure creates two new triangles, for which we must perform the same test, and possibly fix.

Definition 5.11. A triangulation is *locally Delaunay* if for each triangle, any vertex that is opposite to each of the triangle's edges satisfies the empty-circle condition. A triangulation is *globally Delaunay* if every triangle satisfies the empty-circle condition.

Theorem 5.12. *If a triangulation is locally Delaunay, then it is globally Delaunay.*

The intuition behind Theorem 5.12 is that if there were a site p that violated the circumcircle of some triangle Δabc, then there would have to be a vertex of Δabc that violated some triangle Δpxy's circumcircle. Hence the triangulation would not be locally Delaunay.

Note that the cost of insertion is proportional to the degree of the new vertex after it is inserted. Using the Euler characteristic for planar graphs ($V - E + F = 2$), we know that the average vertex degree in the triangulation is less than 6. Therefore, each insertion is done in amortized constant time. Now suppose that we have a Delaunay triangulation. Any of its sites was equally likely to be the last one inserted. Thus, using backwards analysis, we get that the total number of structural changes (which have a cost in $\Theta(1)$) is in $O(n)$.

The last point that we must consider is how we find which triangle a newly added site falls into. One way to do this is by using buckets for each triangle currently in the triangulation. We keep the yet-to-be-inserted sites in their appropriate buckets. Using backwards analysis, we can show that the rebucketing required when we retriangulate an area of the triangulation is done in $O(\log n)$ expected time, for a total running time in $O(n \log n)$.

Chapter 6

Randomized Incremental Algorithms

We have now seen a number of algorithms that we described as randomized incremental. There is a number of things that were common to them all, and this allows us to abstract out the main concept.

Definition 6.1. A *configuration space* is a four-tuple (X, Π, D, K) where

- X is the input set;
- Π is the set of elements that we will call *configurations*;
- $D \subset \mathcal{P}(X)$, and for $\Delta \in \Pi$, we say that $D(\Delta)$ *defines* the configuration Δ;
- $K \subset \mathcal{P}(X)$, and for $\Delta \in \Pi$, we say that $K(\Delta)$ is *in conflict with* or *kills* Δ.

We also have that

$$d \triangleq \max_{\Delta}\{|D(\Delta)| : \Delta \in \Pi\},$$

called the *maximum degree* of the configuration space, is constant, and $D(\Delta)$ and $K(\Delta)$ are disjoint for all configurations $\Delta \in \Pi$. We call a configuration *active* over a set $S \subset X$ if $D(\Delta) \subset S$ and $K(\Delta) \cap S = \emptyset$. Let $\mathcal{T}(S)$ denote the set of active configurations over S.

For a configuration space as defined above, the goal of a randomized incremental algorithm is to compute $\mathcal{T}(X)$. We will redefine the problems that we have seen so far in these terms.

Halfplane Intersection

Here X is a set of halfplanes. Each set $D(\Delta)$ represents a pair of intersecting halfplanes that define an intersecting point Δ, and each set $K(\Delta)$ contains all the halfplanes that do not contain Δ. Therefore, $\mathcal{T}(X)$ is the set of vertices of the common intersection of the halfplanes of X.

Trapezoidal Maps

For X a set of line segments, the configurations in Π are the trapezoids in all of the trapezoidal maps defined by subsets of X. The set $D(\Delta)$ contains the line segments that define Δ. The set $K(\Delta)$ contains the line segments that intersect Δ. Then $\mathcal{T}(X)$ is the set of trapezoids in the trapezoidal map of X.

Delaunay Triangulations

The input here is a set of points in the plane. Each $\Delta \in \Pi$ is a triangle formed by three sites in X. The set $D(\Delta)$ represents the vertices of Δ, and the killing set $K(\Delta)$ represents the points that are in the circumcircle of Δ. The set $\mathcal{T}(S)$ represents the triangles in the Delaunay triangulation of the points in S.

6.1 Computing $\mathcal{T}(X)$

We can now present an abstraction of the procedures that we have been defining so far. Let $x_1, x_2, \ldots, x_n$ be a random permutation of the elements of X. Let $X_r = \{x_1, x_2, \ldots, x_r\}$ be the order in which we added the first r elements while maintaining $\mathcal{T}(X_r)$.

Theorem 6.2. *Let (X, Π, D, K) be a configuration space. Then the expected number of structural changes in any step of a randomized incremental algorithm, is*

$$|\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r-1})| \leq \frac{d}{r} \mathbb{E} |\mathcal{T}(X_r)|.$$

Proof. Not surprisingly, we will show this using backwards analysis. Suppose that we fix X_r to be some set of r elements of X, and that we remove one element from it at random. There is a $\frac{1}{r}$ probability for any of the elements to be chosen. We want to know how many configurations in $\mathcal{T}(X_r)$ are removed when we remove an element. The cost of removing element x is equal to $|\{\Delta \in \mathcal{T}(X_r) : x \in D(\Delta)\}|$. Thus we get

$$\sum_{x \in X_r} |\{\Delta \in \mathcal{T}(X_r) : x \in D(\Delta)\}| \leq d |\mathcal{T}(X_r)|.$$

It follows that the expected cost of removing an element at random from X_r is $\frac{d}{r} |\mathcal{T}(X_r)|$. Letting $|\mathcal{T}(X_r)|$ be a random variable (rather than using a fixed X_r as we've been doing), we get the bound $\frac{d}{r} \mathbb{E} |\mathcal{T}(X_r)|$. ■

Theorem 6.3. *For (X, Π, D, K) a configuration space,*

$$\mathbb{E}\left[\sum_{\Delta} |K(\Delta)|\right] = \sum_{r=1}^{n} d^2 \left(\frac{n-r}{r}\right) \left(\frac{\mathbb{E}|\mathcal{T}(X_r)|}{r}\right).$$

Proof. First, note that

$$\sum_{\Delta} |K(\Delta)| = \sum_{r=1}^{n} \sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)|.$$

Now, let

$$k_y(X_r) \triangleq |\{\Delta \in \mathcal{T}(X_r) : y \in K(\Delta)\}|,$$

and

$$k_{y,x_r}(X_r) \triangleq |\{\Delta \in \mathcal{T}(X_r) : y \in K(\Delta), x_r \in D(\Delta)\}|.$$

We can see that

$$\sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)| = \sum_{y \in X \backslash X_r} k_{y,x_r}(X_r). \tag{6.1.1}$$

If we fix X_r, then the expected value of (6.1.1) depends only on x_r. Since

$$\mathbb{P}[y \in D(\Delta) : \Delta \in \mathcal{T}(X_r)] \leq \frac{d}{r},$$

we have that

$$\mathbb{E} k_{y,x_r}(X_r) \leq \frac{d}{r} k_y(X_r).$$

Thus, by (6.1.1),

$$\mathbb{E}\left[\sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)|\right] \leq \frac{d}{r} \sum_{y \in X \backslash X_r} k_y(X_r). \tag{6.1.2}$$

Observe that each $y \in X \backslash X_r$ is equally likely to be x_{r+1}, so

$$\mathbb{E}k_{x_r+1}(X_r) = \frac{1}{n-r} \sum_{y \in X \backslash X_r} k_y(X_r). \tag{6.1.3}$$

Combining (6.1.2) and (6.1.3), we get

$$\mathbb{E}\left[\sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)|\right] \leq d\left(\frac{n-r}{r}\right) \mathbb{E}k_{x_{r+1}}(X_r). \tag{6.1.4}$$

Now note that $k_{x_{r+1}}(X_r)$ is the number of configurations that will be killed when x_{r+1} is inserted. That is,

$$k_{x_{r+1}}(X_r) = |\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r+1})|. \tag{6.1.5}$$

Putting (6.1.4) and (6.1.5) together, and taking the average over all choices of X_r, we note that the equation holds for all permutations of X. If we sum over all r, we can rewrite (6.1.4) this way:

$$\sum_{r=1}^{n} d\left(\frac{n-r}{r}\right) |\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r+1})| = \sum_{\Delta} d\left(\frac{n-[j(\Delta)-1]}{j(\Delta)-1}\right), \tag{6.1.6}$$

where $j(\Delta)$ is the stage when Δ is killed. For $i(\Delta)$ the stage when Δ is created, we get

$$\frac{n-[j(\Delta)-1]}{j(\Delta)-1} = \frac{n}{j(\Delta)-1} - 1 \leq \frac{n}{i(\Delta)} - 1 = \frac{n-i(\Delta)}{i(\Delta)}, \tag{6.1.7}$$

since $i(\Delta) \leq j(\Delta)-1$. Replacing (6.1.7) in (6.1.6), we have

$$\sum_{r=1}^{n} d\left(\frac{n-r}{r}\right) |\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r+1})| \leq \sum_{\Delta} d\left(\frac{n-i(\Delta)}{i(\Delta)}\right).$$

Now,

$$\sum_{\Delta} d\left(\frac{n-i(\Delta)}{i(\Delta)}\right) \leq \sum_{r=1}^{n} d\left(\frac{n-r}{r}\right) |\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r-1})|,$$

therefore,

$$\mathbb{E}\sum_{r=1}^{n} \sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)| \leq \sum_{r=1}^{n} d\left(\frac{n-r}{r}\right) \mathbb{E}|\mathcal{T}(X_r) \backslash \mathcal{T}(X_{r-1})|.$$

Using Theorem 6.2, we get the result

$$\mathbb{E}\sum_{r=1}^{n} \sum_{\Delta \in \mathcal{T}_r \backslash \mathcal{T}_{r-1}} |K(\Delta)| \leq \sum_{r=1}^{n} d\left(\frac{n-r}{r}\right) \frac{d}{r} \mathbb{E}|\mathcal{T}(X_r)|,$$

as was to be shown. ■

Whew.

6.1.1 Application to Trapezoidal Maps

We can now use configuration spaces to analyze the randomized incremental algorithm provided earlier for point location in trapezoidal maps.

Let X be the set of line segments of the input, Π be the set of trapezoids created during the construction of the map, $D(\Delta)$ be the set of line segments that define a trapezoid Δ, and $K(\Delta)$ be the set of line segments that destroy Δ when inserted. Since each trapezoid is defined by at most 4 line segments, we have that $d = 4$.

We get

$$\begin{aligned}\sum_{\Delta} |K(\Delta)| &= \sum_{r=1}^{n} 16 \left(\frac{n-r}{r}\right)\left(\frac{\mathbb{E}|\mathcal{T}(X_r)|}{r}\right) \\ &\le 16n \sum_{r=1}^{n} \frac{1}{r^2}\mathbb{E}|\mathcal{T}(X_r)| \\ &\le 16n \sum_{r=1}^{n} \frac{cr}{r^2} && \text{For some constant } c \text{ (see Section 4.1).} \\ &= 16cn \sum_{r=1}^{n} \frac{1}{r} \le 16cn(\ln n + 1) \\ &\in O(n \lg n).\end{aligned}$$

6.1.2 Application to Delaunay Triangulations

Let X be the set of sites of the input. Let Π be the set of all triangles defined by the sites of X. Let $D(\Delta)$ be the three points that define triangle Δ, and let $K(\Delta)$ be the set of points that are in the circumcircle of Δ. The set $\mathcal{T}(S)$ is the Delaunay triangulation of the sites of $S \subset X$. Assuming that no four points are cocircular, we have that $d = 3$. Thus, using Theorem 6.3, we have

$$\begin{aligned}\sum_{\Delta} |K(\Delta)| &= \sum_{r=1}^{n} 9 \left(\frac{n-r}{r}\right)\left(\frac{\mathbb{E}|\mathcal{T}(X_r)|}{r}\right) \\ &= 9 \sum_{r=1}^{n} \left(\frac{n-r}{r}\right)\left(\frac{r-2}{r}\right) \\ &\le 9 \sum_{r=1}^{n} \frac{n}{r}\frac{r}{r} \\ &= 9n \sum_{r=1}^{n} \frac{1}{r} \\ &\in O(n \lg n).\end{aligned}$$

Chapter 7

Line Arrangements

We now look at the next most important structure in computational geometry.

Definition 7.1. A *line arrangement*, denoted by $\mathcal{A}(L)$, is the subdivision of the plane defined by a set L of lines. Each line intersection corresponds to a vertex, each line segement between vertices corresponds to an edge, and each region bounded by edges corresponds to a face.

We add a vertex at infinity for infinite edges to reach, in order to have a proper planar graph.

7.1 Point-Line Duality

Proposition 7.2. *Two convex polygons cannot intersect in more than $O(n)$ pairs.*

Proof. Let K_1 and K_2 be convex polygons, with $|K_1| + |K_2| = n$, where we define $|K|$ to be the number of sides of K. Suppose that one side of K_1 intersects three or more sides of K_2. Then K_2 cannot be convex, a contradiction. Therefore, each side of K_1 intersects at most two sides of K_2. It follows that the two polygons intersect in at most $O(n)$ pairs. ■

Proposition 7.3. *Suppose that we have a function $f : (a,b) \mapsto \{(x,y) : ax + by - 1 = 0\}$. For a point p, let $p^* \triangleq f(p)$ be a line. Let $f^{-1}(L) = L^*$ be a point. If p lies on the origin-side of a line L, then L^* lies on the origin side of p^*.*

Proof. Let $p = (p_x, p_y)$ and L be the line described by $ax + by - 1 = 0$. Assume that b is positive. Let p lie on the origin-side of L. Then $ap_x + bp_y - 1 \leq 0$. Given that the dual of this is $p_x a + p_y b - 1 \leq 0$, we see that the result holds simply by the commutativity of multiplication. ■

Proposition 7.4. *Let there be a mapping such that a point $p = (a,b)$ in what we call the primal plane corresponds to a line p^* described by $y = ax - b$ in what we call the dual plane. Suppose that two lines ℓ_1 and ℓ_2 intersect at a point (α, β) in the primal plane. Then the line that passes through ℓ_1^* and ℓ_2^* in the dual plane is defined by $y = \alpha x - \beta$.*

Proof. Let ℓ_1 and ℓ_2 be described by $y = a_1 x - b_1$ and $y = a_2 x - b_2$, respectively. We know that $\beta = a_1\alpha - b_1 = a_2\alpha - b_2$. Hence $\alpha = \frac{b_2 - b_1}{a_2 - a_1}$. Now the line that goes through (a_1, b_1) and (a_2, b_2) in the dual plane is

$$\begin{aligned} y &= \left(\frac{b_2 - b_1}{a_2 - a_1}\right) x - c \\ &= \alpha x - c. \end{aligned}$$

If we substitute x and y by a_1 and b_1, we get

$$\begin{aligned} b_1 &= \alpha a_1 - c \\ c &= \alpha a_1 - b_1 \\ &= \beta. \end{aligned}$$

Therefore, the line that passes through ℓ_1^* and ℓ_2^* in the dual plane is defined by $y = \alpha x - \beta$. ■

7.2 The Zone Theorem

Definition 7.5. Given a line ℓ, the zone of ℓ in $\mathcal{A}(L)$, denoted $Z_{\mathcal{A}}(\ell)$, is the set of faces that have an edge in ℓ.

Theorem 7.6 (Zone Theorem). *Given an arrangement $\mathcal{A}(L)$, the total number of edges in the cells of $Z_{\mathcal{A}}(\ell)$ is at most $6n$, where n is the number of lines in L.*

7.3 Topological Plane Sweep

Several problems involving line arrangements use sweep line algorithms in their solution; the plane is swept by a straight line and events are encountered that trigger certain behaviour. Such algorithms use two data structures over the course of their existence: the sweep line status (typically an array of size n), and a priority queue of size in $O(\log n)$. The latter can be omitted by using a *topological sweep line.* That is, a sweep line that we do not require to be straight.

Definition 7.7. Let $\mathcal{A}(H)$ be an arrangement of lines H. We define a *cut* $\langle c_1, \ldots, c_n \rangle$ to be a list of edges of $\mathcal{A}(H)$ such that c_1 is an edge of the topmost region of the arrangement, and c_n is an edge of the bottommost region. In addition, each elements c_i and c_{i+1}, for $1 \leq i \leq n-1$, are incident to region R_i such that c_i is above and c_{i+1} is below.

We also define a *left-of* relationship. A cut A is left of a cut B if for every $\ell \in \mathcal{A}(H)$, the edge of A on ℓ is the same as, or to the left of, the edge of B on ℓ.

Thus we define the sweep line with *edges* of the arrangements, not lines as we did before.

Definition 7.8. We call an *elementary step* the event where the topological line sweeps past a vertex of the arrangement. There are $\binom{n}{2}$ such events.

Lemma 7.9. *There always exists an elementary step that moves the topological sweep line to the right, unless the cut is the rightmost cut.*

Proof. Note that a cut is righmost if and only if its edges are unbounded in the right direction. Thus if a cut is not rightmost, then there must be some vertex to its right. ■

Definition 7.10. The *upper horizon tree* $T^+(C)$ of a cut C is the tree defined by taking each edge of the cut and extending it to the right until it reaches another edge, at which point only the edge with the higher slope continues to be extended. The *lower horizon tree* $T^-(C)$ is defined symmetrically, where at each vertex, the line with the lower slope is extended. For both types of trees, we add a vertical line at $x = +\infty$ so that the tree is indeed a tree, and not a forest.

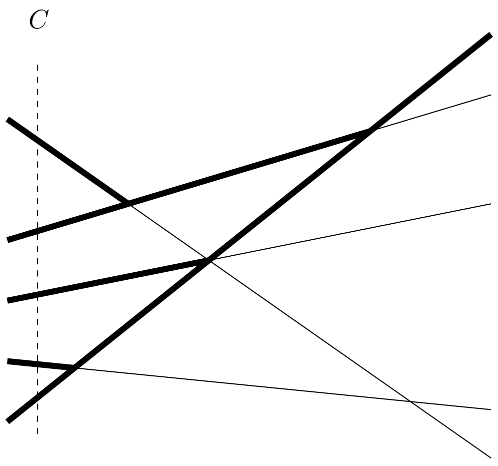


Figure 7.1: The upper horizon tree of a cut C. The dotted line crosses the edges in C. (Note that the cut is not leftmost.)

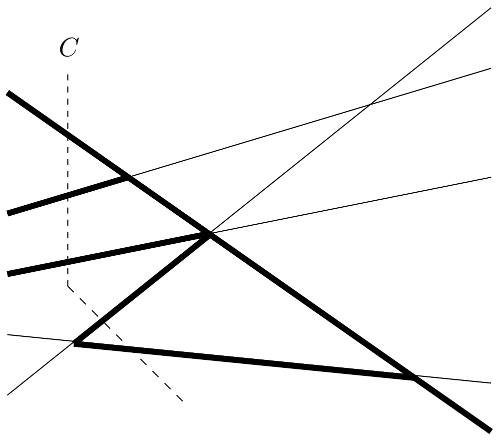


Figure 7.2: The lower horizon tree of the same arrangement with a different cut.

The algorithm proceeds as follows. We begin by building the upper horizon tree for the leftmost cut by insterting the lines in decreasing slope order. Thus the first line segment is going to consist of the entire line with the highest slope, and each subsequent addition will correspond to the infinite halfline that ends when it encounters a line that is already in the horizon tree. We initialize the lower horizon tree similarly. With these two structures defined, we can derive the edges of the leftmost cut in linear time. Thus, given the arrangement in order of line slopes, we initialize the problem in $O(n)$ time. We can also define a stack with the indices i of lines c_i and c_{i+1} that share a right endpoint. This allows us to know where elementary steps happen.

When such an event is encountered, we need to update the upper and lower horizon trees, at a cost in $O(n)$ each. However, since there are $\binom{n}{2} \in O(n^2)$ elementary steps, this gives a running time in $O(n^3)$. We want to have each update in $\Theta(1)$ amortized cost.

Lemma 7.11. *Given an arrangement $\mathcal{A}(H)$ of n lines, the total number of edges traversed in the upper or lower horizon tree through all elementary steps on a line of H is in $O(n)$.*

Proof. Suppose that we are considering a line $\ell \in \mathcal{A}(H)$ and that we are at an elementary step V. What lies below ℓ at this point is a sequence of intersecting lines in increasing order of slope magnitude. Let X be the vertex that has a tangent line parallel to ℓ. Using the accounting method, for each line intersecting below ℓ we will charge one payment to the line that precedes it if the line is left of X, and charge one payment to the line that succedes it otherwise. For the first and last lines in the sequence, note that they intersect ℓ at the current region defined by the intersection of the halfplane below ℓ and the halfplanes above the line sequence. We will charge the two lines at their elementary step with ℓ. Since there are $n-1$ elementary steps along ℓ, this is the amount that will be charged in total for ℓ. ■

Lemma 7.11 implies that the cost of updating the upper and lower horizon trees is in $O(n^2)$ total each. It follows that the topological sweep can be performed in $O(n^2)$ time. We note that the data structures used to carry this out are all in $O(n)$, resulting in $O(n)$ space usage for the algorithm.

Appendix A

Convex Analysis

A.1 Convex Sets

Proposition A.1. *Let $C \subset \mathbb{R}^n$ be a convex set, with $x_1, \ldots, x_k \in C$, and let $\theta_1, \ldots, \theta_k \in \mathbb{R}$ satisfy $\theta_i \geq 0$, $\sum \theta_i = 1$. Then $\theta_1 x_1 + \cdots + \theta_k x_k \in C$.*

Proof. We proceed by induction. Since we know that the case $k = 2$ is true, we have a base case. Thus assume that the membership holds for some arbitrary k. We must show that this implies membership for $k+1$ components. Let $x = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_k x_x + \theta_{k+1} x_{k+1}$. Let $x' = \frac{\theta_1}{\sum_{i=1}^k \theta_i} x_1 + \cdots + \frac{\theta_k}{\sum_{i=1}^k \theta_i} x_k$. Then, by the inductive hypothesis, $x' \in C$. Multiplying both sides of the previous equation by the denominator of the RHS, we get $\theta_1 x_1 + \cdots + \theta_k x_k = \sum_{i=1}^j \theta_i x'$. Given the definition for x, it follows that

$$\begin{aligned} x &= \sum_{i=1}^k \theta_i x' + \theta_{k+1} x_{k+1} \\ &= (1 - \theta_{k+1}) x' + \theta_{k+1} x_{k+1}. \end{aligned}$$

Since C is convex, $x \in C$. ■

Proposition A.2. *A set is convex if and only if its intersection with any line is convex.*

Proof. ($\Rightarrow$) Suppose that a set C is convex, and that ℓ is a line such that $C \cap \ell \neq \emptyset$. Let $x_1, x_2 \in C \cap \ell$. Then, for $\theta \in [0,1]$, $\theta x_1 + (1-\theta) x_2 \in C$ because C is convex. Furthermore, $\theta x_1 + (1-\theta) x_2 \in \ell$, by the definition of a line. Therefore $\theta x_1 + (1-\theta) x_2 \in C \cap \ell$. Thus $C \cap \ell$ is convex.

($\Leftarrow$) Now let $C \cap \ell$ be convex for any ℓ. Then for any two points $x_1, x_2 \in C \cap \ell$ and any $\theta \in [0,1]$, $\theta x_1 + (1-\theta) x_2 \in C \cap \ell$. Since this is true for any ℓ, the membership holds for every x_1 and x_2 in C. Consequently, C is convex. ■

Definition A.3. A set C is *midpoint convex* if whenever two points $a, b \in C$, the average, or midpoint, $(a+b)/2$ is in C.

Proposition A.4. *If C is closed and midpoint convex, then C is convex.*

Proof. Let C be closed an midpoint convex. Then for points $a_0, a_1 \in C$, define $a_2 \triangleq \frac{a_0 + a_1}{2} \in C$. We can thus construct a sequence $\{a_n\}$ where $a_n = \frac{a_{n-1} + a_{n-2}}{2}$. Since C is closed, the sequence converges to a point $a \in C$. We can do this in any manner we want for the points between a_0 and a_1. In other words, the line between a_0 and a_1 is in C. By the previous exercise, this is sufficient to show that C is convex. ■

Proposition A.5. *The convex hull of a set S is the intersection of all convex sets that contain S.*

Proof. Let C be the intersection of all the convex sets that contain S. Let S' be the convex hull of S. We know that the intersection of convex sets is convex. Therefore, since S' is the smallest convex set that contains S, $S' \subset C$.

Now, let $x_1, x_2 \in C$. Then x_1, x_2 are in a convex set, which contains S. Therefore x_1, x_2 are in S', and thus $C \subset S'$. ■

Proposition A.6. *The distance between two parallel hyperplanes* $\mathbf{a}^T\mathbf{x} = b_1$ *and* $\mathbf{a}^T\mathbf{x} = b_2$ *is equal to* $\frac{|b_1 - b_2|}{\|\mathbf{a}\|_2}$.

Proposition A.7. *A halfspace* $\mathbf{a}_1^T\mathbf{x} < b_1$ *is a subset of another halfspace* $\mathbf{a}_2^T\mathbf{x} < b_2$ *if and only if* $\mathbf{a}_1 = \lambda\mathbf{a}_2$ *and* $b_1 < \lambda b_2$, *for some* $\lambda \in \mathbb{R}$.

Proposition A.8. *Let* a, b *be distinct points in* $\mathbb{R}^n$. *Then the set of all points that are closer to* $\mathbf{a}$ *than* $\mathbf{b}$ *using the Euclidean norm is a halfspace.*

Proof. Let $\{\mathbf{x} : \|\mathbf{x} - \mathbf{a}\|_2 \le \|\mathbf{x} - \mathbf{b}\|_2\}$ be such a set of points. Let $\mathbf{c} = 2(\mathbf{b} - \mathbf{a})$ and $d = \|\mathbf{b}\|_2^2 - \|\mathbf{a}\|_2^2$. We have

$$\begin{aligned}
\|\mathbf{x} - \mathbf{a}\|_2 &\le \|\mathbf{x} - \mathbf{b}\|_2 \\
\sqrt{(x_1 - a_1)^2 + \cdots + (x_n - a_n)^2} &\le \sqrt{(x_1 - b_1)^2 + \cdots + (x_n - b_n)^2} \\
(x_1 - a_1)^2 + \cdots + (x_n - a_n)^2 &\le (x_1 - b_1)^2 + \cdots + (x_n - b_n)^2 \\
(-2)a_1x_1 + a_1^2 + \cdots + (-2)a_nx_n + a_n^2 &\le (-2)b_1x_a + b_1^2 + \cdots + (-2)b_nx_n + b_n^2 \\
(-2)\mathbf{a}^T\mathbf{x} + \|\mathbf{a}\|_2^2 &\le (-2)\mathbf{b}^T\mathbf{x} + \|\mathbf{b}\|_2^2 \\
2(\mathbf{b} - \mathbf{a})^T\mathbf{x} &\le \|\mathbf{b}\|_2^2 - \|\mathbf{a}\|_2^2 \\
\mathbf{c}^T\mathbf{x} &\le d,
\end{aligned}$$

which defines a halfspace in $\mathbb{R}^n$. ■

Definition A.9. Let $\mathbf{x}_0, \ldots, \mathbf{x}_K \in \mathbb{R}^n$. Consider the set of points that are closer, in Euclidean distance, to $\mathbf{x}_0$ than the other $\mathbf{x}_i$, that is,

$$V = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{x}_0\|_2 \le \|\mathbf{x} - \mathbf{x}_i\|_2, i = 1, \ldots, K\}.$$

Then V is called the *Voronoi region* around $\mathbf{x}_0$ with respect to $\mathbf{x}_1, \ldots, \mathbf{x}_K$.

Proposition A.10. *A set* V *is a Voronoi region if and only if it is a polyhedron.*

Proof. Note that it follows from Proposition A.8 that Definition A.9 describes the intersection of halfspaces. Therefore any Voronoi region is a convex polyhedron $V = \{A\mathbf{x} \preceq \mathbf{b}\}$, where each row i of A is equal to $2(\mathbf{x}_i - \mathbf{x}_0)$ and each component i of $\mathbf{b}$ is equal to $\|\mathbf{x}_i\|_2^2 - \|\mathbf{x}_0\|_2^2$.

For the converse, suppose that we are given a polyhedron $P = \{A\mathbf{x} \preceq \mathbf{b}\}$ with nonempty interior. Let $\mathbf{x}_0$ be any point in the interior of P, and let each $\mathbf{x}_i$ be the point such that $\mathbf{x}_0$ and $\mathbf{x}_i$ form a line whose middle point crosses hyperplane $P_i = \{\mathbf{x} : \mathbf{a}_i^T\mathbf{x} = b_i\}$ perpendicularly. That is, the distance $\|\mathbf{x}_0 - \mathbf{x}_i\|_2$ is equal to $2\|\mathbf{x}_0 - P_i(\mathbf{x}_0)\|_2 = 2\|\mathbf{x}_i - P_i(\mathbf{x}_i)\|_2$, where we define $P_i(\mathbf{y})$ to be the point $\mathbf{x} \in P_i$ that minimizes $\|\mathbf{x} - \mathbf{y}\|_2$. Then each hyperplane divides the space into two regions where points are closer to $\mathbf{x}_i$ or closer to $\mathbf{x}_0$, depending on which side of the hyperplane they fall on. It follows that P is the Voronoi region of $\mathbf{x}_0$. ■

Appendix B

General Topology

B.1 Topological Spaces

Definition B.1. A *topological space* is a pair (X, τ), where X is a set and τ is a collection of subsets of X, that satisfies three axioms:

(i) $\emptyset, X \in \tau$.

(ii) The set τ is closed under union.

(iii) The set τ is closed under finite intersection.

We say that members of τ are *open*, and we call τ a *topology* on X. The pair $(X, \mathcal{P}(X))$, where $\mathcal{P}(X)$ is the power set of X, is called the *discrete topology*; the pair $(X, \{\emptyset, X\})$ is called the *trivial topology*. Given two topologies τ and τ', if $\tau \subset \tau'$, then we say that τ' is *finer* than τ, and τ is *coarser* than τ'.

We will sometimes write a topological space X to mean a topological space (X, τ).

Proposition B.2. *Every open set in $\mathbb{R}$ is the union of a collection of disjoint open intervals (a, b), where we allow $a = -\infty$ and $b = \infty$.*

Proof. Let O be an open set in $\mathbb{R}$. Then for any $x \in O$, there is an interval $(a, b) \subset O$ that contains x. If, for any $x, y \in O$, there are open intervals (a, b) and (c, d) (with $a < b$, $c < d$) in O that contain x and y respectively, such that $(a, b) \cap (c, d) \neq \emptyset$, we can take $(a, b) \cup (c, d)$ to be an open interval containing x and y. Thus any open set in $\mathbb{R}$ is the union of a collection of disjoint open intervals. ■

Proposition B.3. *Let $\mathcal{O}$ be the collection of all intervals $I_a = (a, \infty)$ in $\mathbb{R}$, including the cases $I_\infty = \emptyset$ and $I_{-\infty} = \mathbb{R}$. Then $\mathcal{O}$ defines a topology on $\mathbb{R}$ and the closure of a set $A \subset \mathbb{R}$ is a half-closed interval of the form $[a, \infty)$.*

Proof. First, we are given that $\emptyset, \mathbb{R} \in \mathcal{O}$, thus the first axiom is satisfied. Now, for any two intervals I_a and I_b, we have that $I_a \cup I_b = I_c$, where $c = \min\{a, b\}$. Thus $\mathcal{O}$ is closed under unions. Finally, $I_a \cap I_b = I_c$, where $c = \max\{a, b\}$. Hence $\mathcal{O}$ is closed under finite intersection. Therefore, $\mathcal{O}$ defines a topology on $\mathbb{R}$. ■

Definition B.4. A subset A of a topological space X is *closed* if its complement $X \backslash A$ is open.

Proposition B.5. *For a subset A of a topological space X:*

(a) Every open set contained in A is contained in A°. That is, A° is the largest open set contained in A.

(b) Every closed set containing A contains $\overline{A}$. That is, $\overline{A}$ is the smallest closed set containing A.

Proof.

(a) Assume that A is not open, otherwise the result is trivial. Let O be an open subset of A, and let $x \in O$. Since O is open, there is a set $O' \subset O$ such that $x \in O'$. Now, $O' \subset A$, so x is an interior point of A.

(b) Assume that A is open. Let C be a closed set containing A. Let x be a limit point of A. Then $x \in \overline{A}$. Since x is a limit point of A, x is a limit point of C. Because C is closed, $x \in C$. Hence $\overline{A} \subset C$.

■

Corollary B.6. *If A is closed, then $A = \overline{A}$. If A is open, then $A = A^\circ$.*

Proposition B.7. *For arbitrary subsets A, B of a topological space X,*

(a) $\overline{A \cup B} = \overline{A} \cup \overline{B}$;

(b) $\overline{A \cap B} \subset \overline{A} \cap \overline{B}$;

(c) $(A \cap B)^\circ = A^\circ \cap B^\circ$;

(d) $(A \cup B)^\circ \supset A^\circ \cup B^\circ$.

Proof.

(a) We have $A \subset \overline{A}$ and $B \subset \overline{B}$. Thus $A \cup B \subset \overline{A} \cup \overline{B}$. It follows that $\overline{A \cup B} \subset \overline{A} \cup \overline{B}$, since the right hand side is closed and, by Proposition B.5(b), it must contain the smallest closed set that contains $A \cup B$.

For the converse, $A \cup B \subset \overline{A \cup B}$. Then $\overline{A} \subset \overline{A \cup B}$ and $\overline{B} \subset \overline{A \cup B}$ (Proposition B.5(b)). Thus $\overline{A} \cup \overline{B} \subset \overline{A \cup B}$.

(b) $A \cap B \subset \overline{A}$ and $A \cap B \subset \overline{B}$. Thus $A \cap B \subset \overline{A} \cap \overline{B}$. It follows again from Proposition B.5(b) that $\overline{A \cap B} \subset \overline{A} \cap \overline{B}$.

(c) We have $A^\circ \cap B^\circ \subset A$ and $A^\circ \cap B^\circ \subset B$. Hence $A^\circ \cap B^\circ \subset A \cap B$. The result follows from Proposition B.5(a).

(d) $A^\circ \subset A \cup B$ and $B^\circ \subset A \cup B$, thus $A^\circ \cup B^\circ \subset A \cup B$. The result follows again from Proposition B.5(a).

■

Example B.8. *Here are examples where equality fails to hold in Proposition B.7 (b) and (d).*

Suppose that $X = \mathbb{R}$, $A = \mathbb{Q}$, and $B = \mathbb{R} \backslash \mathbb{Q}$. Then $\overline{A} = \overline{B} = \mathbb{R}$. It follows that $\overline{A} \cap \overline{B} = \mathbb{R}$, but $\overline{A \cap B} = \emptyset$.

Similarly, with A and B as above, $(A \cup B)^\circ = \mathbb{R}$, but $A^\circ \cup B^\circ = \emptyset \cup \emptyset = \emptyset$.

Proposition B.9. *If A is a subset of a topological space X, then*

(a) $\overline{X \backslash A} = X \backslash A^\circ$;

(b) $(X \backslash A)^\circ = X \backslash \overline{A}$.

Proof. (a) For A open, $\overline{X \backslash A} = X \backslash A = X \backslash A^\circ$, using Corollary B.6. Thus assume that A is not open. If A is neither open nor closed, then $A^\circ = \emptyset$ and $\overline{X \backslash A} = X = X \backslash A^\circ$.

For A closed, let $x \in \overline{X \backslash A} = \partial(X \backslash A) \cup (X \backslash A)^\circ = \partial A \cup X \backslash A \subset$. TODO

For the converse, note that $X \backslash A^\circ = \overline{X \backslash A^\circ} = \partial(X \backslash A^\circ) \cup (X \backslash A^\circ)^\circ$. TODO

(b) We have that $(X \backslash A)^\circ$ is open. Thus, for $x \in (X \backslash A)^\circ$, x is not in A, and x is in the interior of X so x is not in the closure of A. Thus $x \in X \backslash \overline{A}$.

For the converse, let $x \in X \backslash \overline{A}$. Thus $x \in X$ and $x \notin \overline{A}$. It follows that $x \notin A$, that is, $x \in X \backslash A$. Since $X \backslash \overline{A}$ is open, x is an interior point. Therefore, $x \in (X \backslash A)^\circ$, by Proposition B.5(a). ■

Proposition B.10. *The boundary ∂A always contains $\partial(A^\circ)$.*

Proof. Let $x \in \partial(A^\circ)$. Then every open set O with $x \in O$ has points in both A° and $X\backslash A^\circ$. We need to show that O also has points in both A and $X\backslash A$. Since $A^\circ \subset A$, it is clear that O has points in A. Now, using Proposition B.9, we have that O has points in both A and $\overline{X\backslash A}$. If A is open, we are done, since by Proposition B.5(b), $\overline{X\backslash A} = X\backslash A$. For A closed, note that $\overline{X\backslash A} = \partial(X\backslash A) \cup (X\backslash A)^\circ$. Also note that $\partial(X\backslash A) = \partial A$. Thus we must show that O has points in $(X\backslash A)^\circ$. But since A is closed, $X\backslash A$ is open, and by Proposition B.5(a), it is equal to $(X\backslash A)^\circ$. This proves the result. ■

Definition B.11. Let X, Y be topological spaces, and let $O \subset Y$ be an open set. A function $f : X \to Y$ is said to be *continuous* if $f^{-1}(O)$ is open.

Definition B.12. Let (X, τ) be a topological space. A collection $\mathcal{B} \subset \tau$ is called a *basis* for τ if every set in τ is a union of sets in $\mathcal{B}$.

We now give a useful characterization of Definition B.12.

Proposition B.13. *Given a topological space (X, τ), $\mathcal{B}$ is a basis for τ if and only if:*

(i) for all $x \in X$, there exists a set $B \in \mathcal{B}$ such that $x \in B$;

(ii) for all pairs $B_1, B_2 \in \mathcal{B}$, and for all $x \in B_1 \cap B_2$, there is a set $B_3 \in \mathcal{B}$ such that $x \in B_3 \subset B_1 \cap B_2$.

Now we can use Proposition B.13 to prove that a collection forms a basis for a topological space.

Definition B.14. A *neighbourhood* of a point x in a topological space X is any set $A \subset X$ that contains an open set O containing x.

Definition B.15. A *metric* on a set X is a function $d : X \times X \to \mathbb{R}$ such that

(i) $d(x, y) \geq 0$, with equality holding only when $x = y$;

(ii) $d(x, y) = d(y, x)$;

(iii) $d(x, y) \leq d(x, z) + d(z, y)$. This is called the *triangle inequality.*

Proposition B.16. *The metric topology on a subset A of a metric space X is the same as the subspace topology.*

Lemma B.17. *For a subspace $A \subset X$ that is open (closed) in X, a subset $B \subset A$ is open (closed) in A if and only if it is open (closed) in X.*

Lemma B.18. *Given a space X, a subspace Y, and a subset $A \subset Y$, then the closure of A in the space Y is the intersection of the closure of A in X with Y.*

Definition B.19. A continuous map $f : X \to Y$ is a *homeomorphism* if it is a bijection, and its inverse function $f^{-1} : X \to Y$ is also continuous.

Example B.20. *The function f : donut $\to$ coffee mug is a homeomorphism.*

B.2 Connectedness

Definition B.21. A space X is *connected* if it cannot be decomposed as the union of two disjoint nonempty open sets.

Theorem B.22. *An interval $[a, b]$ in $\mathbb{R}$ is connected.*

Definition B.23. A space X is *path-connected* if for each pari of points $a, b \in X$, there exists a path in X from a to b, that is, a continuous map $f : [0, 1] \to X$ with $f(0) = a$ and $f(1) = b$.

Proposition B.24. *If a space is path-connected, then it is connected.*

Proposition B.25. *Suppose that $f : X \to Y$ is continuous and onto. Then:*

1. *If X is connected, then so is Y.*
2. *If X is path-connected, then so is Y.*

Lemma B.26. *For a point $x \in X$, let $P(x)$ be the subspace of X consisting of points z such that there is a path in X from x to z. If $P(x) \cap P(y) \neq \emptyset$, then $P(x) = P(y)$.*

Lemma B.27. *If a subspace A of a space X is connected, then so is $\overline{A}$.*

Lemma B.28. *If A is a subset of a space X that is both open and closed, then any connected subspace $C \subset X$ that meets A must be contained in A.*

Proposition B.29. *If $\{C_\alpha\}$ is a family of connected subspaces of a space X, any two of which have nonempty intersection, then $\bigcup\limits_{\alpha} C_\alpha$ is connected.*

Proposition B.30. *If each point in a space X has a neighbourhood that is path-connected, then the path components of X are also the connected components.*

References

[1] David M. Mount, Course Notes for CMSC 754, University of Maryland, Spring 2012. `http://www.cs.umd.edu/class/spring2012/cmsc754/`

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational Geometry, Algorithms and Applications*, Third Edition, Springer, 2008.

[3] David G. Kirkpatrick and Raimund Seidel, *The Ultimate Planar Convex Hull Algorithm?*, SIAM J. Comput., 15-1 (1986), pp. 287-299.

[4] Avraham A. Melkman, *On-Line Construction of the Convex Hull of a Simple Polyline*, Information Processing Letters, 25 (1987), pp. 11-12.

[5] J. Michael Steele and Andrew C. Yao, *Lower Bounds for Algebraic Decision Trees*, Journal of Algorithms, 3 (1982), pp. 1-8.

[6] Michael Ben-Or, *Lower Bounds for Algebraic Computation Trees*, ACM (1983), pp. 80-86.

[7] Raimund Seidel, *Small-Dimensional Linear Programming and Convex Hulls Made Easy*

[8] Jack Snoeyink, *Handbook of Discrete and Computational Geometry*, Chapter 27, Chapman & Hall/CRC.

[9] Herbert Edelsbrunner and Leonidas J. Guibas, *Topologically Sweeping an Arrangement*, DEC Systems Research Center (1986).

[10] Steven Boyd and Lieven Vandenberghe, *Convex Optimization*, Cambridge University Press, NY, 2004.

[11] Allen Hatcher, Course Notes on Introductory Topology, Cornell University, 2005. `http://www.math.cornell.edu/~hatcher/Top/Topdownloads.html`